PATENT
Docket No.: 81924.0002
Express Mail Label No.: EL715813464US

ANALYTICAL VIRTUAL MACHINE

PRIORITY CLAIM AND RELATED APPLICATION

This application claims priority from U.S. provisional patent application Serial No. 60/242,939 filed October 24, 2000, which application is incorporated by reference in its entirety. This application is related to U.S. patent application Serial No. 09/642,625, filed August 18, 2000, which application is incorporated by reference in its entirety.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a virtual machine system and, more particularly, relates to a virtual machine system appropriate for automated code analysis and capable of analyzing data including executable programs presented to a computer system.

2. Discussion of the Related Art

Detection of malicious code including programs such as viruses has been a concern throughout the era of the personal computer. With the growth of communication networks such as the Internet and increasing interchange of data, including the rapid growth in the use of e-mail for communications, the infection of computers through communications or file exchange is an increasingly significant consideration. Infections take various forms, but are typically related to computer viruses, trojan programs, or other forms of malicious code. Recent incidents of e-mail mediated virus attacks have been dramatic both for the speed of propagation and for the extent of damage, with Internet service providers (ISPs) and companies suffering service problems and a loss of e-mail capability. In many instances, attempts to adequately prevent file exchange or e-mail mediated infections significantly inconvenience computer users. Improved strategies for detecting and dealing with virus attacks are desired.

One conventional technique for detecting viruses is signature scanning. Signature scanning systems use sample code patterns extracted from known malicious code and scan for the occurrence of these patterns in other program code. In some cases program code that is scanned is first decrypted through emulation, and the resulting code is scanned for signatures or function signatures. A primary limitation of this signature scanning method is that only known malicious code is detected, that is, only code that matches the stored sample signatures of known malicious code is identified as being infected. All viruses or malicious code not previously identified and all viruses or malicious code created after the last update to the signature database will not be detected. Thus, newly created viruses are not detected by this method; neither are viruses with code in which the signature, previously extracted and contained in the signature database, has been overwritten.

In addition, the signature analysis technique fails to identify the presence of a virus if the signature is not aligned in the code in the expected fashion. Alternately, the authors of a virus may obscure the identity of the virus by opcode substitution or by inserting dummy or random code into virus functions. Nonsense code can be inserted that alters the signature of the virus to a sufficient extent as to be undetectable by a signature scanning program, without diminishing the ability of the virus to propagate and deliver its payload.

Another virus detection strategy is integrity checking. Integrity checking systems extract a code sample from known, benign application program code. The code sample is stored, together with information from the program file such as the executable program header and the file length, as well as the creation date and creation time for the program file. The program file is checked at regular intervals against this database to ensure that the program file has not been modified. A main disadvantage of an integrity check based virus detection system is that a great many warnings of virus activity issue when any modification of an application program is performed. For example, integrity checking programs generate long lists of modified files when a user upgrades the operating system of the computer or installs or upgrades application software. It is difficult for a user to determine when a warning represents a legitimate attack on the computer system.

Checksum monitoring systems detect viruses by generating a cyclic redundancy check (CRC) value for each program file. Modification of the program file is detected by a variation in the CRC value. Checksum monitors improve on integrity check systems in that it is more difficult for malicious code to defeat the monitoring. On the other hand, checksum monitors exhibit the same limitations as integrity checking systems in that many false warnings issue and it is difficult to identify which warnings represent actual viruses or infection.

Behavior interception systems detect virus activity by interacting with the operating system of the target computer and monitoring for potentially malicious behavior. When such malicious behavior is detected, the action is blocked and the user is informed that a potentially dangerous action is about to take place. The potentially malicious code can be allowed to perform this action by the user. This makes the behavior interception system somewhat unreliable, because the effectiveness of the system depends on user input. In addition, resident behavior interception systems are sometimes detected and disabled by malicious code.

Another conventional strategy for detecting infections is the use of bait files. This strategy is typically used in combination with other virus detection strategies to detect an existing and active infection. This means that the malicious code is presently running on the target computer and is modifying files. The virus is detected when the bait file is modified. Many viruses are aware of bait files and do not modify files that are either too small, obviously bait files because of their structure or have a predetermined content in the file name.

It is apparent that improved techniques for detecting viruses and other malicious types of code are desirable.

Aspects of the present invention utilize certain characteristics of virtual machine technology. The concept of a "virtual machine" is known in the art and virtual machines have found various uses. The merits of the "virtual machine" include the ability to execute code that would not execute on the hardware platform under other circumstances, such as code intended for other hardware platforms. Other applications of virtual machine technology can be found in multi-user and multi-processing systems, where each process runs within its own virtual machine.

Virtual machines have been applied to various computer functions, such as in the interface between computer hardware and high level languages (HLL) (U.S. Patent No. 5,872,978 to Hoskins), the networking of real machines to form a parallel processor (U.S. Patent No. 5,774,727 to Walsh et al.) and to create a multi-tasking or multi-user computer environment (U.S. Patent No. 4,400,769, to Kaneda et al.). Virtual machines have also been applied where cross-platform HLL code portability is required (U.S. Patent No. 6,118,940 to Alexander, III et al).

SUMMARY OF THE PREFERRED EMBODIMENTS

An aspect of the invention provides a virtual machine system for computer code behavior analysis, the virtual machine system having a software processor.

The processor stores a behavior record including behavior flags representative of computer code behavior observed by virtually executing the computer code under analysis within the virtual machine. A sequencer stores a sequence in which behavior flags are set in the behavior record during virtual execution of the computer code under analysis. Simulated memory and a simulated operating system representative of a host real computer system are provided and the computer code under analysis interacts with the simulated memory and the simulated operating system to generate the behavior flags. The virtual machine passes data representative of the behavior record to the host real computer system prior to termination of the virtual machine.

Another aspect of the present invention provides a virtual machine system for computer code behavior analysis having a software processor. The virtual machine includes a register or structure that stores behavior flags representative of computer code behavior observed by virtually executing the computer code under analysis within the virtual machine. The virtual machine also includes a register or structure that stores a sequence in which behavior flags are set in the behavior flags register or structure. Registers or structures store all entry points to the computer code under analysis within the virtual machine. A structure stores interrupt vector addresses that point at interrupt service routines loaded into memory reserved by the virtual machine when the virtual machine is initialized. A memory structure simulates input and output ports and another memory structure simulates processor memory. One or more operating system simulation shells simulate values returned by a real operating system under which the computer code under analysis is intended to operate.

BRIEF DESCRIPTION OF DRAWINGS

FIG. 1 illustrates a configuration of an analytical virtual machine with DOS MZ-type executable or binary file.

FIG. 2 illustrates a configuration of an analytical virtual machine with high level language (HLL) program code.

FIG. 3 illustrates a configuration of an analytical virtual machine running PE, NE or LE Windows executable code.

FIG. 4 shows a memory map of an analytical virtual machine running a binary (COM or SYS) executable and running a Visual Basic (VB) executable.

FIGS. 5A and 5B schematically illustrate a table listing behavior pattern, sequencer and entry point structures generated by a preferred implementation of the analytical virtual machine.

FIG. 6 shows a schematic procedure flow diagram of a preferred implementation of the analytical virtual machine executing binary machine code.

FIG. 7 shows a schematic procedure flow diagram of a preferred implementation of the analytical virtual machine executing HLL.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Preferred implementations of the present invention provide an analytical virtual machine (AVM) system that executes program code within an emulated computer system in a manner similar to how the code would be executed in a real computer system. Preferred implementations of an analytical virtual machine do not allow physical input or output to take place or any interaction between the program code under analysis and the real or physical computer system. Instead, input and output operations, system calls, and instructions are simulated in a manner transparent to the code under analysis. System functions, operating system application program interface (API) calls, input and output commands and alterations to pre-defined memory locations preferably are all logged by the analytical virtual machine during execution of the code being analyzed. Most preferably the analytical virtual machine returns the logged data to the real or physical computer system as both a behavior pattern and a sequence structure representing the sequence in which the behavior pattern bits were set during virtual execution.

The analytical virtual machine (AVM) described here is intended to be used in automated code function analysis and behavior extraction. The "code" is either binary machine code or high level language (HLL) in either tokenized or source code text format. The analytical virtual machine executes the code to be analyzed from each entry point in the entry point table that is passed to it. While virtually executing code, the AVM monitors system calls, input/output

(I/O) operations and memory accesses. Either machine language instructions or high-level language instructions are executed within the emulated central processing unit (CPU) of the AVM. Operating system functions called by the application code, input/output port read and write operations and memory read and write operations performed by the application code are simulated within the AVM environment.

Previous analytical systems scanned code without virtual execution. In such systems, application code was scanned for function calls or target code snippets (U.S. Patent No. 5,440,723 to Arnold et al.). The scanning method has several disadvantages, the first being that a code pattern match may be out of synchronization with the actual executed code. Another disadvantage of the scanning method is that direct access to a function, that is a function access that is not through a documented call structure, is not detected. In addition, a control fields in memory, which are filled by code execution, are not present and so are not analyzed in the scanning method. A third disadvantage is that persons who write malicious code are likely to disguise the real function of that code by inserting 'do nothing' code into the function template or by calling the function in an unconventional manner. Because of these disadvantages a scanning analysis system is less than exact.

Preferred implementations of the present invention employ an analytical virtual machine like that presented here. Such a virtual machine executes application program code in step with the real program flow beginning from each entry point. When using such a particularly preferred analytical virtual machine, an exact representation of the functions contained within the application program code is obtained. 'Do Nothing' instructions no longer are effective in disguising code. Calling a function in an unconventional manner will have no effect if the AVM is implemented in the particularly preferred manner to respond as the real machine would.

As will be explained below in greater detail, the use of a virtual machine in code analysis has an advantage that, as compared to a conventional native machine, code is analyzed by executing the code within a safe environment. Here "safe" refers to the fact that the operating system, programs and data of the real computer system do not interact with code under analysis and so are not likely to be damaged by the code under analysis. Analysis nevertheless takes place in step with normal code execution, as if the code were executed on a native machine using a real processor system.

Thus, preferred implementations of the analytical virtual machine can accommodate a plurality of operating systems and hardware platforms for simulation. The organization of the computing system can be flexibly modified. Further, it is possible to supervise the operation of the system. In certain presently preferred embodiments, a supervision function is built into the virtual operating system of the AVM to analyze the behavior of the unknown program code. FIG. 1 shows a block diagram of a current implementation of the AVM within a physical computer system. Like a compiler-based virtual machine, the

AVM processes the flow of the code stream. Unlike a compiler-based virtual machine, code is executed and virtual operating system functions are called.

FIG. 1 shows the organization of a computer system running an instance of the AVM that has been initialized for execution of a DOS MZ-type executable or DOS binary COM or SYS program, including the boot sector loader used by such programs. The area inside the box labeled 100 represents a real machine, that is, the hardware of a computer comprising the physical central processing unit (CPU), physical memory, and user input and output channels. The input and output channels include the keyboard, mouse, video display, disk drives and other peripheral equipment connected to the processing unit. The operating system 102 is stored in physical memory 106 together with device drivers and applications programs that are in the process of execution, likely within other virtual machines. All such software is executed by the real central processing unit (CPU) 105. One device driver is shown in the block diagram, which is the file system hook device driver 101. File system hook device driver 101 hooks to the real operating system file system and informs the application which launches the AVM of modifications to files stored on the hard disk drive.

The operating system software 102 resides in memory 106 and operates within the real machine 100. The application program 103 that initializes and receives the results of the analytical virtual machine 200 exists on the real machine, executing within the operating system 102 environment, as does the analytical virtual machine 200. After the analytical virtual machine 200 is initialized, the application program passes the entry point, the file type and a buffer containing the segment holding the program's main entry point through 103 to the program loader pre-processor 211.

Once the virtual machine is created, a part of physical memory 106 is reserved by the application program for use by the virtual machine. This memory block is labeled 210 and exists in physical memory 106. The preprocessor 211 prepares virtual memory block 210 for use by the analytical virtual machine and creates all appropriate, simulated operating system memory blocks, such as the interrupt vector table (IVT) at virtual addresses 0000 to 1023, the DOS parameter area at virtual addresses 1024 to 1279, the program area and the memory reserved for the VGA display (mapped at virtual address 0A000h). The program loader pre-processor 211 creates memory control blocks (MCB's) and then loads the entry point code into this memory. In simulated high memory, above the 640K boundary of the IBM PC base memory map, virtual interrupt service routines are created. All addresses referenced by the program code under analysis are remapped to fit inside this memory model by software memory mapper 207.

After the program loader 211 completes the initialization of the virtual DOS memory model, the virtual CPU 205 commences fetching program instructions through the prefetch mechanism 203. Depending on the second byte (the Or/m byte) of each instruction fetched and decoded by software instruction decoder 204, the software or virtual CPU fetches from the register stack 206 or from the virtual memory 210, using the data fetch mechanism 208 and memory mapper 207.

The software CPU 205 performs the desired operation on the data and writes the results back to the destination determined by the Or/m byte. Referenced operating system functions are simulated in the operating system simulator 201. Interrupt services are simulated in the interrupt services simulator 212, with vectors stored in virtual memory 210 at addresses 0000 to 1023. The interrupt services in high memory act as a link between the interrupt vector table and the simulated interrupt services 212. The memory block 210 also contains the system stack and the application code stack. The system and application code stacks are not shown, since they are created at the addresses contained within the application code under analysis and vary from application to application. Data are fetched from the virtual stack area reserved in memory 210 if the control byte Or/m indicates that this is required.

The software CPU 205 has a prefetch mechanism 203, and an instruction decoder 204. As instructions are fetched, decoded and executed, the program code under analysis performs simulated functions. The execution of each simulated function sets and resets flags in the behavior flags register 209 and the sequence in which these operations take place is recorded in sequencer 213. The resulting behavior flag pattern, together with the sequencer structure, are passed to the application that initiated the analytical virtual machine. The analytical virtual machine is then terminated in presently preferred embodiments.

This process is further illustrated in FIG. 6, which shows a block diagram of a procedure flow within a preferred implementation of the AVM. FIG. 6 is, like the other figures, illustrative of a preferred implementation but is not intended to limit the scope of the present invention. The program loader pre-processor function is show in the top left hand corner. The application program passes the file type, code length, entry point offset and a buffer of length "len" containing the entire entry segment code as well as the SS (stack segment), SP (stack pointer), CS (code segment) and the IP (instruction pointer) register values contained within the executable file. The loader pre-processor then reserves a block of physical memory for use by the AVM as virtual memory. The size of the memory block reserved for virtual memory is at this moment limited to 64000h bytes (409600 decimal). However, the size of the virtual memory is not relevant to the operation of the AVM and so should not be considered limiting.

The virtual memory map is initialized to contain the interrupt vector table, the BIOS parameter area, the DOS parameter area, the environment string table, the program segment prefix (PSP) and the display adapter memory block (mapped at 0A000h). An area is reserved above the display adapter memory block in which interrupt service routines (ISR's) are created for each of the 1024 interrupts contained within the interrupt vector table (IVT). Then the remaining memory is configured as memory blocks controlled by memory control blocks (MCB's). The entry segment code, passed to the AVM by the controlling application program, is placed within the MCB memory blocks in virtual memory. Next, the virtual processor's segment registers; SS, ES, DS, CS, GS and FS, are initialized with values that depend on the type of executable code that is placed in virtual memory. Then the register stack EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI and the flags register are initialized. The monitor function, which sets and resets flags in the behavior register, is built into individual operating system calls, interrupt calls and API calls, as well as the address remapper and the interrupt service routines. During each of the following instruction processing loops, the AVM checks if the maximum allowed CPU time has been exceeded. This virtual CPU time limit is most preferably imposed to break deadlock conditions, where the code under analysis causes the AVM to enter an endless loop. Proper configuration of the CPU time limit allows long decryption or polymorphic loops to be processed with risking deadlock.

The IP register points to the entry point in the virtual memory. At this point in the virtual execution process the virtual CPU starts to fill the 12-byte prefetch queue. The bytes are decoded according to Intel's Pentium instruction set reference, whereby the first byte in this 12-byte queue determines the function of the instruction word. The instruction word comprises an operation, an interrupt call or an operating system API call. Of course, in implementations for other processors, other instruction set references will be appropriate and the specific details will vary. It will be apparent from this discussion how to implement the analytical virtual machine for different processors and different operating system implementations.

In case of an operation, the FetchData procedure is called, which retrieves the correct data either from virtual memory, the prefetch queue or from the processor's registers, depending on the values stored in the 2nd byte of the prefetch queue (the Or/m byte). The data thus retrieved is processed, e.g. added, divided, multiplied or processed through a multitude of other arithmetic or logic operators applied to the data. Then the SetFlags procedure is called, which evaluates the result of the operation that was performed and sets flags in the virtual flags register accordingly. The SetSign procedure adjusts the sign of the processed results. The MemRemap procedure takes as its input the address of the destination in virtual memory. This address is remapped to fit into the 409600 bytes reserved for virtual memory. Monitoring of address space modifications is performed in the MemRemap procedure.

In case of an interrupt call, the call index is passed to a procedure that passes control to the appropriate interrupt service routine by looking up the address in the interrupt vector table (IVT). Analysis then continues by executing the appropriate interrupt service routine, either in virtual BIOS, virtual DOS, virtual DPMI, virtual Windows (native API) or in the code under analysis if that code has modified the IVT to point at one of its own procedures. Monitoring of interrupt functions is performed in the appropriate procedures.

In case of an API call, the call is passed to a procedure that attaches the relevant ordinal number to the call, and passes control to the virtual API. This virtual API contains procedures that simulate the response, but not the functionality, of the real operating system API. Monitoring of API functions is performed in each API procedure. API functions modify areas of virtual memory, so that subsequent virtual API calls can read back the correct and expected results.

This sequence of events continues to take place until a terminate program system call is encountered or a far jump is performed that is outside the current segment. Whenever a branch instruction is encountered, the AVM will use the parameters intended by the original programmer to jump, but store the other side of the branch in the entry point table together with the conditions used in the branch instruction. The entry point table is traversed and code is executed from each entry point in the entry point table. No duplication is allowed in the entry point table. When all entry points have been processed, the resulting behavior pattern is returned to the calling application, together with the sequencer structure and the AVM is terminated, releasing the reserved memory block. The calling program within the real computer system can then review the behavior pattern and the sequencer to evaluate the code analyzed by the AVM.

FIG. 2 shows a configuration of the AVM with a code interpreter shell loaded, of the type appropriate to running a high level language. The program loader 301 splits the program code into individual lines, indexes the program lines and places them in virtual memory 310. During this load operation, entry point information is extracted from the code and placed in an entry point table, together with the program line index that each entry point references. Variables used in the program code are extracted in the AVM's second pass through the program code. Variables are stored in a variable structure in virtual memory 310, containing the variable index, the variable type, and the current value of the variable, initialized to zero prior to emulation. At this point the code interpreter 305 receives control and starts to fetch instructions from the first entry point in the entry point table. Instructions are interpreted, operating on the variables in the variable structure at the position from which instructions are fetched (jump instructions) or they call system services (open file, write file etc.) and perform application program interface (API) calls. System services are simulated in the services simulation procedure 312. The Calc. Next procedure 308 calculates the next line number in memory 310 from which the next instruction line is to be fetched by the Fetch Next procedure 307. The behavior flags register 302 monitors the initialization of certain system variables, system service and API calls. These events set and reset bits in the behavior flags register 302. The sequence in which behavior register flags are set or reset is also recorded in the sequencer structure 309. Scratchpad storage 306 is used to temporarily store the condition of system services, which may be referenced at a later time by the application program code.

The system performs this sequence of events for each instruction line, and continues at each entry point in the code until all entry points in the entry point table have been processed. The AVM then returns the behavior pattern, together with the sequencer structure, to the calling program and the AVM is terminated. The process flow of the FIG. 2 instance of the AVM is further illustrated in FIG. 7, which shows the processing flow within the current AVM implementation after the AVM has been initialized to interpret and extract the behavior pattern of a high level language (HLL) program, such as VB Script code. The language definition key-word list and rules define the language interpreted by the processing core. In FIG. 7, a keyword list is shown that is a cross between VBA and VBS, but the processing core is not limited to those languages.

When the AVM of FIG. 7 is initialized, a 409600 byte (hexadecimal 0x64000) block of real memory is reserved as virtual computer memory. The loader function initializes the memory and then loads the code buffer passed to it into virtual memory as individual lines, marked by a carriage return (CR = 0Dh) or a CR and line feed (CR LF = 0D 0Ah) character codes. These lines are indexed into a source structure, whereby each line is assigned an address. Processing the source lines commences in pass 1, whereby all entry points defined in the code, such as menu entries, auto-executing procedures and procedures attached to standard system functions are placed in an entry-point table. In pass 2, all variables that are present in the code are extracted and placed in a variable structure within virtual memory. Each variable is stored as an address, the variable type, the variable name and the variable value.

After this the AVM processing core begins fetching instruction lines from the first entry point to the code. The line is decoded using the keyword list and the language rules. Variables references by the code line are fetched from the variable structure and processed. The variable value is then written back to the appropriate position. In case of an API call to perform an operating system function, the virtual API is referenced and it looks up the internal ordinal number of the API function, performs the simulated function and returns the simulated call values back to the HLL calling function. During execution the process flow causes flags in the behavior register 302 (FIG. 2) to be set or reset, and the sequence of these events to be stored in a sequencer structure 309 (FIG. 2). This process continues until all entry points in the entry point table have been processed, at which time the allocated memory is released, the behavior register value and the sequencer structure are returned to the application program for analysis, and the AVM is terminated.

In a present implementation, an analytical virtual machine in accordance with the present invention has three operating modes, a high level mode, a protected mode and a real mode. The real mode corresponds to an operating system such as DOS and the corresponding instance of the AVM is illustrated in FIGS. 1 and 6. The high level mode might be used to analyze a program in a high level language such as Perl, Visual Basic, or a scripting language. A high level mode of the AVM is illustrated in FIGS. 2 and 7.

A protected mode instance of the AVM might be used to analyze thirty-two bit Windows code or Linux code. An illustration of the configuration of the AVM for protected mode use is illustrated in, for example, FIG. 3. The structure and operation of the FIG. 3 AVM is apparent from that illustration and the detailed discussion of the high level and real mode analytical virtual machines and so is not discussed further here. The primary difference as compared to the real mode implementation is the kernel and API functionality of the protected mode system. These aspects of the protected mode reflect the fact that it is already running a virtual machine and the fact that API calls are made rather than interrupts.

FIG. 4 illustrates the various memory maps that are used by the analytical virtual machine in the different modes identified here. As shown, the programs to be analyzed by the different mode devices load differently. Consequently, the analytical virtual machine is configured especially for each of these different modes and determines where the program to analyze is to begin execution and how the program should be analyzed. The memory allocation illustrated here is exemplary and is subject to optimization for particular systems. As such, future developments for future processors and programs are expected to alter the precise characteristics of the AVM and its implementations illustrated here.

The end product of an invocation of an AVM in accordance with preferred embodiments of the present invention are the contents of the behavior flag register and the sequencer. FIGS. 5A and 5B illustrate an exemplary and presently preferred set of behaviors that are tracked to characterize the code under analysis. This list of behaviors is presently preferred because it accurately tracks potentially malicious behavior and precisely characterizes presently contemplated forms of code. It is anticipated that future, different behavior sets or modifications of the illustrated behavior set might be desirable. The sequencer is a data structure that tracks the evolution of the behavior flag register. The sequence of setting the flags in the behavior register is particularly significant in analyzing and fully characterizing analyzed code. The sequencer is a data structure whose size is adapted to accurately characterize the code. Both the end pattern in the behavior register and the sequence of flag settings and resettings stored in the sequence are passed to the real machine just prior to terminating the virtual machine. The calling application program then uses the data in these structures to characterize the analyzed code.

The preceding discussion described a virtual machine that performs analysis of an application program (code) within a protected execution environment on a real computer. This analytical virtual machine (AVM) comprises a pre-processor which creates, in the memory reserved by the virtual machine, an image of the appropriate operating system under which the application software program is intended to execute. The AVM identifies the operating system and configures the execution environment by the file format and control fields within the header of the file that stores the application program and by the program code to be analyzed. The AVM is run by the operating system of the real computer to execute the application program contained within the AVM. No direct interaction is allowed to exist between the application program and the system software execution environment and / or the computer hardware.

An AVM is created by the computer immune system application for each analysis and is destroyed when that analysis is complete. The AVM is constructed out of a number of layered shells. The configuration of AVM shells depends on the format of the application program that needs to be analyzed; e.g. a software CPU shell is loaded in case native program code is analyzed, while in the case of high level language script or program code the appropriate language interpreter is loaded. Therefore, the processor core of the AVM exists either as a CPU executing native code or as a high level language interpreter. Operating system calls that are contained within the application software program are simulated in such a way that the application program appears to execute within a physical computer environment.

The application program is executed in several passes through the AVM, depending on the structure of the application software program, and may not be executed in any sequence as intended by the original creator of the application program. The aim of execution within the AVM is to perform an analysis that extracts the program code behavior under every condition contained within that program. Once this aim has been satisfied, the analytical virtual machine is terminated, preserving the generated behavior pattern and the sequencer structure, which contains the sequence in which events recorded in the behavior pattern have taken place.

The AVM described here is well suited for and is intended for use with the computer immune system and method described in U.S. patent application Serial No. 09/642,625 filed August 18, 2000. Application Serial No. 09/642,625 is incorporated by reference in its entirety as disclosing further aspects of a preferred implementation and application of the described analytical virtual machine.